Advanced Operating Systems

Through tracing, analysis, and experimentation

ACS/Part III L41: Advanced Operating Systems
Part II: Advanced Operating Systems

Lecture 1, Part 1: What is an Operating System?

Prof. Robert N. M. Watson

2022-2023

MPhil/Part III L41 vs. Part II AdvOpSys

- These lectures are shared by two separate courses:
  - ACS / Part III L41: Advanced Operating Systems
  - Part II: Advanced Operating Systems
- The two courses also share an online lab framework based on the RPi4, JupyterLabs, DTrace, and HWPMC
  - Lab 1 is a tutorial shared by the two courses
- But there are some important differences:
  - Key difference 1: Assessed coursework (after lab 1)
    - L41 has **2x** independently written **lab reports**
    - Part II has **2x** short-answer **lab assignments**
  - Key difference 2: Assigned readings
    - L41 assigns additional research readings
- Please be sure to use the right material for your course!

Getting started

- What is an operating system?

Lecture 1, Part 1

- About the module
- Systems research
- Lab assignments / reports

Lecture 1, Part 2

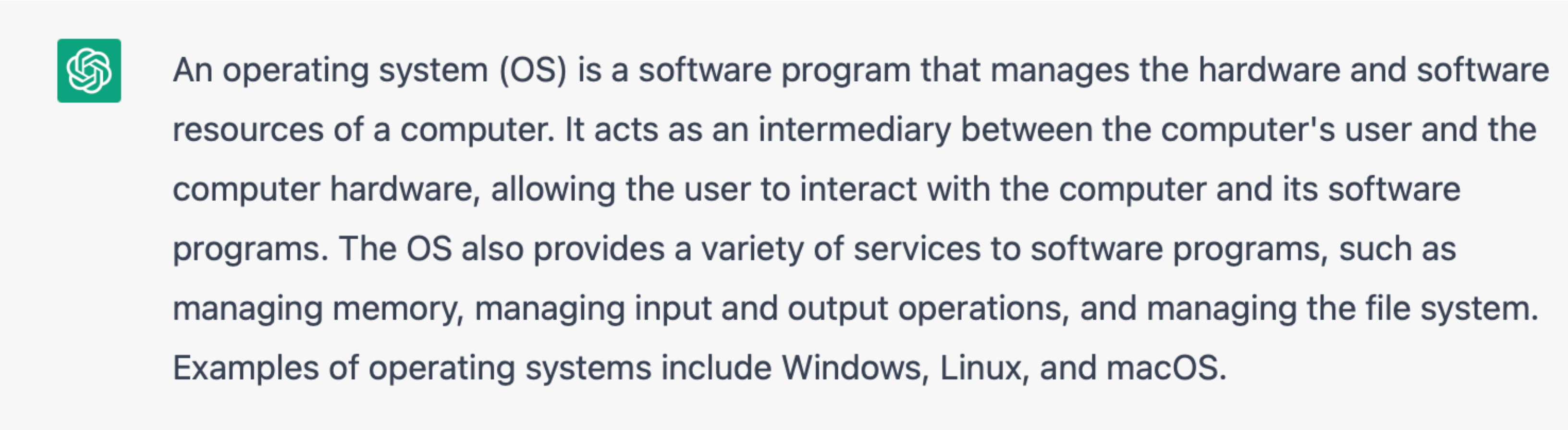
- Kernel dynamics
- Readings for next lecture

Lecture 1, Part 3

What is an operating system?

> [An OS is] low-level software that supports a computer's basic functions, such as scheduling tasks and controlling peripherals.
>
> - Google

What is an operating system?



An operating system (OS) is a software program that manages the hardware and software resources of a computer. It acts as an intermediary between the computer's user and the computer hardware, allowing the user to interact with the computer and its software programs. The OS also provides a variety of services to software programs, such as managing memory, managing input and output operations, and managing the file system. Examples of operating systems include Windows, Linux, and macOS.

- ChatGPT

What is an operating system?

But that is basically the 1970s definition,
and not at all a contemporary one.

Today's general-purpose operating systems consist of
GB of binaries and hundreds of millions of LoC.

Further, when you select an operating system,
you select hardware and software ecosystems.

What is an operating system?

Access control?

Local file systems?

User authentication?

Distributed file-system clients and servers?

Virtual machines?

Multimedia?

Threads and processes?

Networking and WiFi?

Kernel and userspace?

Remote management?

Run-time linker?

Backup?

Debug and trace?

Application packaging?

Shell and command-line tools?

Device drivers?

System libraries?

Language runtimes?

Crypto libraries?

Secure enclaves?

Web browser?

Class libraries?

Remote access?

Payment services?

Software updates?

Window system?

Profiling and optimization?

Crashdump collection

.. And surely lots more

General-purpose operating systems

… are for **general-purpose computers:**

- Servers, workstations, mobile devices
- Run **applications** – i.e., software unknown at OS design time
- Abstract the hardware, provide services, ‘class libraries’
- E.g., Windows, Apple macOS, Android, iOS, Linux, BSD, …

Userspace	Local and remote shells, GUI, management tools, daemons Run-time linker, system libraries, logging and tracing facilities
– system-call layer –	
Kernel	System calls, hypercalls, remote procedure call (RPC)* Processes, filesystems, IPC, sockets, management Drivers, packets/blocks, protocols, tracing, virtualisation VM, malloc, linker, scheduler, threads, timers, tasks, locks

* Continuing disagreement on whether distributed-filesystem servers and window systems ‘belong’ in userspace or the kernel

Other kinds of operating systems (1/3)

Specialise the OS for a specific application or environment:

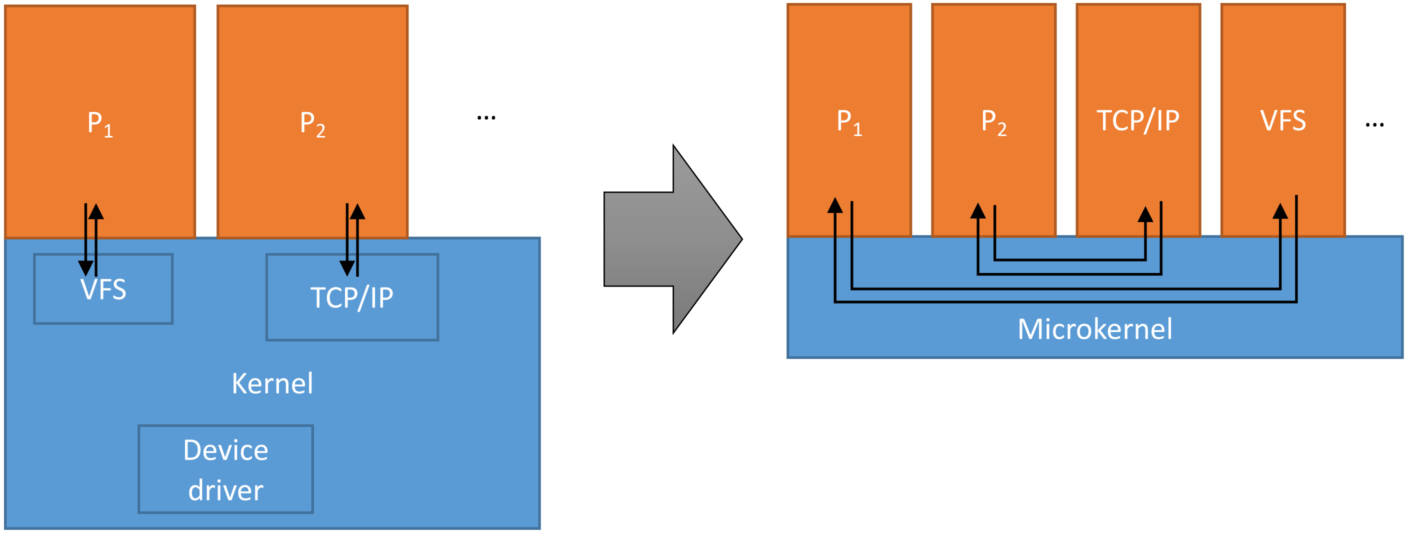
- **Embedded, real-time operating systems**
  - Serve a single application in a specific context
    - E.g., WiFi access points, medical devices, washing machines, cars
  - Small code footprint, real-time scheduling
  - Might have virtual memory / process model
  - Microkernels or single-address space: VxWorks, RTEMS, L4
  - Now also: Linux, BSD (sometimes over a real-time kernel), etc.
- **Appliance operating systems**
  - Apply embedded model to higher-level devices/applications
  - File storage appliances, routers, firewalls, ...
    - E.g., Juniper JunOS, Cisco IOS, NetApp OnTap, EMC/Isilon
  - Under the hood, almost always Linux, BSD, etc.

Key concept: **Operating system as a reusable component**

Other kinds of operating systems? (2/3)

What if we rearrange the boxes?

- **Microkernels, library operating systems, unikernels**
  - Shift code from kernel into userspace to reduce Trusted Computing Base (TCB); improve robustness/flexibility; ‘bare-metal’ apps
  - Early 1990s: Microkernels are king!
  - Late 1990s: Microkernels are too slow!
    - (But ideas about OS modularity dating from this period are widespread)
  - 2000s/2010s: Microkernels are back! But now ‘hypervisors’
  - Sometimes: programming-language runtime as OS



Other kinds of operating systems? (3/3)

- **Hypervisors**
  - Kernels host processes; hypervisors host virtual machines
    - Type-1: Standalone hypervisors (e.g., Xen)
    - Type-2: Integrated with OS kernel (e.g., KVM)
  - Virtualised hardware interface rather than POSIX APIs
  - Paravirtualisation reintroduces OS-like APIs for performance
  - E.g., System/370, VMware, Xen, KVM, VirtualBox, bhyve, Hafnium, ...
  - Many microkernel ideas have found a home here
- **Containers**
  - Hosts multiple userspace instances over a common kernel
  - Controlled namespaces prevent inappropriate accesses
  - Really more about code/ABI (Application Binary Interface) distribution and maintenance

What does an operating system do?

- Key hardware-software surface (w/compiler toolchain)
- Low-level abstractions and services
    - **Operational model**: bootstrap, shutdown, watchdogs
    - **Process model, IPC**: processes, threads, IPC, program model
    - **Resource sharing**: scheduling, multiplexing, virtualisation
    - **I/O**: drivers, local/distributed filesystems, network stack
    - **Security**: authentication, encryption, ACLs, MAC, audit
    - **Local or remote access**: console, window system, SSH
    - **Libraries**: math, protocols, RPC, crypto, UI, multimedia
    - **Monitoring/debugging**: logs, profiling, tracing, debugging

Compiler? Text editor? E-mail package? Web browser?
Can an operating system be “distributed”?

Advanced Operating Systems

Through tracing, analysis, and experimentation

ACS/Part III L41: Advanced Operating Systems
Part II: Advanced Operating Systems

Lecture 1, Part 2: The Course

Prof. Robert N. M. Watson

2022-2023

Why study operating systems?

The OS plays a central role in **whole-system design** when building efficient, effective, and secure systems:

- Strong influence on whole-system performance
- Critical foundation for computer security
- Exciting programming techniques, algorithms, problems
  - Virtual memory; network stack; filesystem; run-time linker; …
- Co-evolves with platforms, applications, users
- Multiple active research communities
- Reusable techniques for building complex systems
- Boatloads of fun (best text adventure ever)

Where is the OS research?

A sub-genre of **systems research**:

- Evolving hardware-software interfaces
  - New computation models/architectures
  - New kinds of peripheral devices
- Integration with programming languages and runtimes
- Concurrent/parallel programming models; scheduling
- Security and virtualisation
- Networking, storage, and distributed systems
- Tracing and debugging techniques
- Formal modeling and verification
- As a platform for other research – e.g., mobile systems

Venues: SOSP, OSDI; ATC; EuroSys; HotOS; FAST; NSDI; HotNets; ASPLOS; USENIX Sec.; ACM CCS; IEEE SSP; …

What are the research questions?

Just a few examples: By changing the OS, can I…

- Create new abstractions for new hardware?
- Make my application run faster by…
  - Better masking latency?
  - Using parallelism more effectively?
  - Exploiting new storage mediums?
  - Adopting distributed-system ideas in local systems?
- Make my application more {reliable, energy efficient}
- Limit {security, privacy} impact of exploited programs?
- Use new language/analysis techniques in new ways?

Systems research focuses on **evaluation** with respect to **applications** or **workloads**: How can we measure whether it is {faster, better, ...}?

Teaching operating systems

- Two common teaching tropes:
  - **Trial by fire**: in micro, recreate classic elements of operating systems: microkernels with processes, filesystems, etc.
  - **Research readings course**: read, present, discuss, and write about classic works in systems research
- This module adopts elements of both styles while:
  - mitigating the risk of OS kernel hacking in a short course
  - working on real-world systems rather than toys; and
  - targeting research skills not just operating-system design
- Trace and analyse real systems driven by specially crafted benchmarks
- Possible only because of (fairly) recent developments in tracing and hardware-based performance analysis tools

Aims of the module (1/2)

Teaching **methodology**, **skills**, and **knowledge** required to understand and perform research on contemporary operating systems by…

- Employing systems methodology and practice
- Exploring real-world systems artefacts through performance and functional evaluation/analysis
- Developing scientific writing skills **(L41 only)**
- Reading original systems research **(L41 only)**

Aims of the module (2/2)

On completion of this module, students should:

- Have an understanding of high-level OS kernel structure.
- Gained insight into hardware-software interactions for compute and I/O.
- Have practical skills in system tracing and performance analysis.
- Have been exposed to research ideas in system structure and behaviour. **(L41 only)**
- Have learned how to write systems-style performance evaluations. **(L41 only)**

Prerequisites

We will take for granted:

- **High-level knowledge of OS terminology** from an undergraduate course (or equivalent); e.g.,:
  - What **schedulers** do
  - What **processes** are … and how they differ from threads
  - What **Inter-Process Communication (IPC**) does
  - How might a simple **filesystem** might work
- Reasonable fluency in **reading** multithreaded C
- Good working knowledge of Python
- Comfort with the UNIX command-line environment
- Undergraduate skills with statistics
  (mean/median/mode/stddev/*t*-tests/linear regression/boxplots/scatterplots ...)

You can pick up some of this as you go (e.g., IPC, Python, or *t*-tests), but will struggle if you are missing several

Module structure – four complementary strands

- **Lectures (✕5: 4 in-person 2-hour slots, 1 prerecorded)**
  - Theory, methodology, architecture, and practice
- **Assigned research and applied readings**
  - Selected portions of module texts – learn skills, methodology
  - Related research readings – research exposure **(L41 only)**
- **In-person lab exercises (✕3 labs, prerecorded lecturelets)**
  - Short prerecorded lecturelet introduces each lab
  - RPi4 cluster to run experiments (one board per student)
  - 6✕ Module demonstrators available to answer questions
- **First lab assignment**
  - Acclimate to platform
  - Learn essential skills to perform later labs (e.g., DTrace, Jupyter)
- **Later lab assignments (Part II – ✕2) or reports (L41 – ✕2)**
  - Based on experiments done in lab exercises
  - Develop scientific + writing skills suitable for systems research (L41)

Outline of module schedule

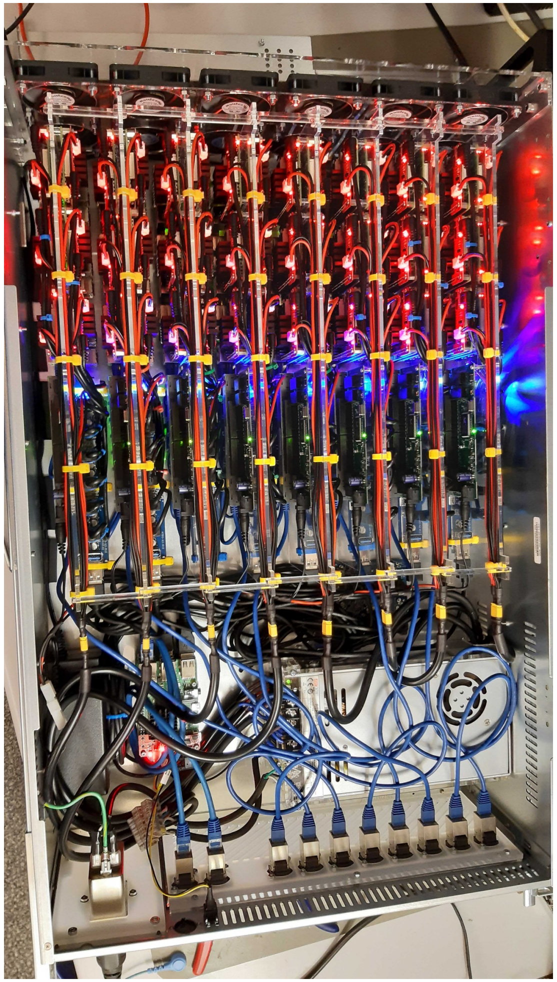
- Submodule 1: **Introduction to kernels and tracing/analysis**
  - 2 lectures (one prerecorded)
  - 2 labs: Introduction to kernel tracing, I/O
  - **Introduction**: OSes, Systems Research, and L41
  - **The Kernel**: Kernel and Tracing

- Submodule 2: **The Process Model**
  - 2 lectures, 2 labs (IPC, PMC)
  - **The Process Model** (1) – Binaries and Processes
  - **The Process Model** (2) – Traps, System Calls, and Virtual Memory

- Submodule 3: **The Network Stack (TCP/IP)**
  - 1 lecture, no lab
  - **The Network Stack** – Sockets, NICs, Work Distribution, and TCP

- Please consult online materials for all deadlines

The lab platform



- 50x Raspberry Pi 4 boards in a rack
  - Broadcom BCM2711 SoC
  - 4x 64-bit A72 ARMv8-A cores
  - 8GB DRAM, 64G SD Card
- FreeBSD operating system
  - DTrace tracing tool
  - HWPMC counter framework
  - Bespoke potted benchmarks motivating OS and microarchitectural performance analysis
  - Jupyter lab notebook environment
- Remotely accessed via SSH + tunneling for Jupyter

Shared first Lab 1:
Getting started with kernel tracing

- Identical assignment for Part II and L41
- Exercises to get you started on the platform; teach:
  - Jupyter Lab Notebooks
  - DTrace instrumentation and data collection – in particular, tracing and profiling scripts
  - Relevant Python plotting tools including Flame Graphs
  - And first dirty hands with respect to OS internals
- Submitted only via Moodle; use “Print to PDF” in your browser to generate a PDF to submit
- Low proportion of marks (10%): really about teaching basic skills you will need for later labs

Lab Assignments 2 and 3 **(Part II only)**

- A series of questions requiring short answers
  - Answers consist of written text, selected data, and plots
  - Perform your work in the Jupyter lab framework
  - Your submission will consist of generated PDF of the completed lab notebook – e.g., by printing to a PDF file
  - Submissions are accepted only via Moodle
- Ensure that your submission is well presented; e.g.,
  - Plots don’t span page boundaries or run off the side
  - Plots have clearly labeled axes, data sets, and so on
  - Make sure your text is concise and clear, addressing the questions that are answered
- Marked based on submitted data, text, and plots
- The third lab assignment (TCP/IP) is optional

Lab Reports 2 and 3 **(L41 only)**

Lab reports document an experiment and analyse its results – typically using **one or more hypotheses**.

Our lab reports will contain the following sections (see notes, template):

1. Title + abstract (1 page)	5. Conclusion (1-2 para)
2. Introduction (1-2 para)	6. References
3. Experimental setup and methodology (1-2 pages)	7. Appendices
4. Results and discussion (3-4 pages)	

Some formats break out (e.g.) experimental setup vs. methodology, and results vs. discussion. The combined format seems to work better for systems experimentation as compared to (e.g.) biology.

- The target length is **8 pages excluding appendices, references**
- **Over-length reports** will be penalized – please stop by the limit!
- **Appendices** will not be read if too long, and should not be essential to understanding the core content of the report

Module texts – core material

You will need to make frequent reference to these books both in the labs and outside of the classroom:

Operating systems: Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson, ***The Design and Implementation of the FreeBSD Operating System, 2nd Edition***, Pearson Education, Boston, MA, USA, September 2014.

Performance measurement: Raj Jain, ***The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling***, Wiley - Interscience, New York, NY, USA, April 1991.

Tracing and profiling: Brendan Gregg and Jim Mauro, ***DTrace: Dynamic Tracing in Oracle Solaris***, Mac OS X and FreeBSD, Prentice Hall Press, Upper Saddle River, NJ, USA, April 2011.

The FreeBSD and DTrace books are available online via vlebooks.com:

https://www.vlebooks.com/Vleweb/Search/Keyword?keyword=freebsd

Module texts – additional material

If your OS recollections feel a bit hazy:

Operating systems: Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. ***Operating System Concepts***, Eighth Edition, John Wiley & Sons, Inc., New York, NY, USA, July 2008.

If you want to learn a bit more about architecture and measurement:

Performance measurement and diagnosis: Brendan Gregg, ***Systems Performance: Enterprise and the Cloud***, Prentice Hall Press, Upper Saddle River, NJ, USA, October 2013.

Advanced Operating Systems

Through tracing, analysis, and experimentation

ACS/Part III L41: Advanced Operating Systems
Part II: Advanced Operating Systems

Lecture 1, Part 3: Kernel dynamics

Prof. Robert N. M. Watson

2022-2023

The kernel: “Just a C program”?

- I claimed that the kernel was mostly “just a C program”
- This is indeed mostly true, especially in higher-level subsystems

Userspace	Kernel
`crt/csu`	`locore`
`rtld`	Kernel linker
Shared objects	Kernel modules
`main()`	`main(), platform_start()`
`libc`	`libkern`
POSIX threads API	`kthread` KPI
POSIX filesystem API	VFS KPI
POSIX sockets API	`socket` KPI
DTrace	DTrace
…	…

The kernel: not just *any* C program

- **Core kernel**: ≈3.4M LoC in ≈6,450 files
    - **Kernel runtime**: Run-time linker, object model, scheduler, memory allocator, threads, debugger, tracing, I/O routines, timekeeping
    - **Base kernel**: VM, process model, IPC, VFS w/20+ filesystems, network stack (IPv4/IPv6, 802.11, ATM, …), crypto framework
    - Includes roughly ≈70K lines of assembly over ≈6 architectures
- Alternative C runtime – e.g., `SYSINIT`, `curthread`
- Highly concurrent – really very, very concurrent
- Virtual memory makes pointers .. odd
- Debugging features – e.g., `WITNESS` lock-order verifier
- **Device drivers**: ≈3.0M LoC in ≈3,500 files
    - 415 device drivers (may support multiple devices)

Spelunking the kernel

```
% ls
Makefile        ddb/            libkern/        nfs/            teken/
amd64/          dev/            mips/           nfsclient/      tests/
arm/            dts/            modules/        nfsserver/      tools/
arm64/          fs/             net/            nlm/            ufs/
bsm/            gdb/            net80211/       ofed/           vm/
cam/            geom/           netgraph/       opencrypto/     x86/
cddl/           gnu/            netinet/        powerpc/        xdr/
compat/         i386/           netinet6/       riscv/          xen/
conf/           isa/            netipsec/       rpc/
contrib/        kern/           netpfil/        security/
crypto/         kgssapi/        netsmb/         sys/

% ls kern
Make.tags.inc           kern_sendfile.c         subr_prng.c
Makefile                kern_sharedpage.c       subr_prof.c
bus_if.m                kern_shutdown.c         subr_rangeset.c
capabilities.conf       kern_sig.c              subr_rman.c
clock_if.m              kern_switch.c           subr_rtc.c
cpufreq_if.m            kern_sx.c               subr_sbuf.c
...
```

- Kernel source lives in `/usr/src/sys`:
    - `kern/` – core kernel features
    - `sys/` – core kernel headers

How work happens in the kernel

- Kernel code executes concurrently in multiple threads
  - User threads in the kernel (e.g., a system call)
  - Shared worker threads (e.g., callouts)
  - Subsystem worker threads (e.g., network-stack workers)
  - Interrupt threads (e.g., Ethernet interrupt handling)
  - Idle threads

```
# procstat -at
  PID    TID COMM             TDNAME           CPU  PRI STATE   WCHAN
    0 100000 kernel           swapper           -1   84 sleep   swapin
    0 100006 kernel           dtrace_taskq      -1   84 sleep   -
...
   10 100002 idle             -                 -1  255 run     -
   11 100003 intr             swi3: vm           0   36 wait    -
   11 100004 intr             swi4: clock (0)   -1   40 wait    -
   11 100005 intr             swi1: netisr 0    -1   28 wait    -
...
   11 100018 intr             intr16: ti_adc0    0   20 wait    -
   11 100019 intr             intr91: ti_wdt0    0   20 wait    -
   11 100020 intr             swi0: uart        -1   24 wait    -
...
  739 100064 login            -                 -1  108 sleep   wait
  740 100079 csh              -                 -1  140 sleep   ttyin
  751 100089 procstat         -                  0  140 run     -
```

Work processing and distribution

- Many operations begin with system calls in a user thread
- But may trigger work in many other threads; for example:
  - Triggering a callback in an interrupt thread when I/O is complete
  - Eventually writing back data to disk from the buffer cache
  - Delayed transmission if TCP isn't able to send immediately
- We will need to be careful about these things, as not all work we are analysing will be in the obvious user thread
- Multiple mechanisms provide this asynchrony; e.g.:

callout	Closure called after wall-clock delay
eventhandler	Closure called for key global events
task	Closure called .. eventually
`SYSINIT`	Function called when module loads/unloads

* Where *closure* in C means: function pointer, opaque data pointer

Wrapping up

- In this lecture, we have:
  - Explored the idea of an operating system
  - Detailed the structure of the course and its expectations
  - The dynamics of kernel execution (just a taster)

- Our next **prerecorded** lecture (intended to be watched before you start on Lab 1) will explore:
  - DTrace, the kernel tracing facility we will use
  - The *probe effect* and its impact
  - Our lab environment

- Readings for the next lecture:
  - Paper - Cantrill, et al. 2004
  - McKusick, et al. Chapter 3 (Kernel Subsystems)