

# Operating Systems: Processes

CIS\*3110: Operating Systems

Dr. A. Hamilton-Wright

School of Computer Science  
University of Guelph

2025-01-28

# Parts of a Computer



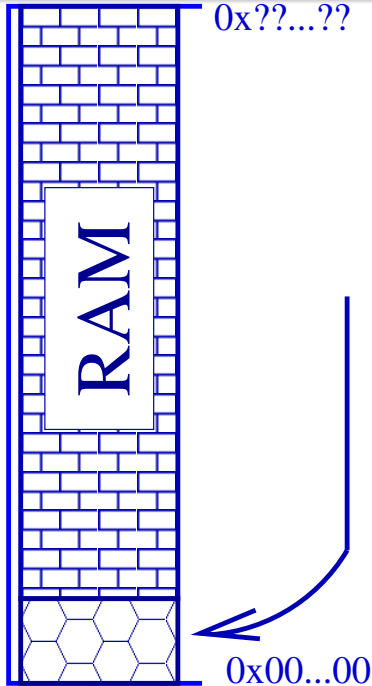
A diagram showing three components of a computer: CPU, RAM, and I/O Devices. The CPU is represented by a blue square on the left. The RAM is represented by a red square in the top right. The I/O Devices are represented by a black square in the bottom right.

CPU

RAM

I/O  
Devices

# Actual Physical Memory



"Zero Page"

- interrupt vector

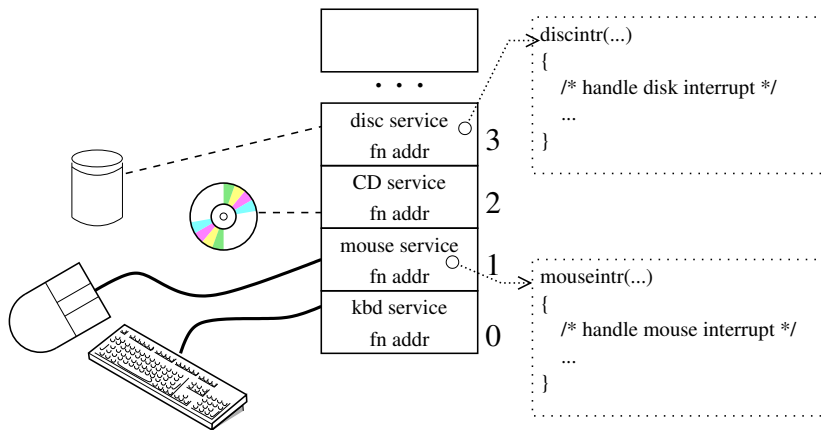
- I/O

BIOS/ROMs (if present)

# Tools at the Hardware Architecture Level

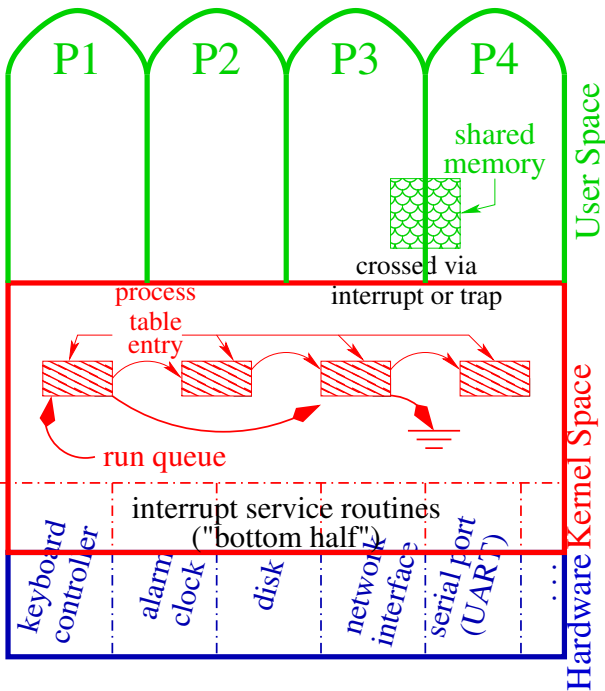
- program counter (PC)
- stack pointer/frame pointer
- processor status register
- general purpose registers
- memory management unit (mapping, protection)
- kernel/user mode
- interrupts and interrupt vector

# Interrupt vector links devices with service routines



- basic **instruction cycle** of the ALU of the CPU:
  1. fetch instruction pointed to by the PC
  2. increment PC
  3. execute instruction

} over and over again
- **processor status** (usually a special register on each CPU)
  - user mode – can execute **most** instructions (but no I/O)
  - kernel mode – can execute **all** user instructions, PLUS access special instructions and registers that control I/O, memory access (also called **supervisor mode**)
- **interrupt vector** (special range of locations in RAM)
  - used to link hardware interrupts with ... **interrupt service routines**



# Program on an Example Machine

<i>Assembler</i>		<i>Mach Code</i>	<i>Virt Addr</i>
	load R1, 0xA2	0x11A2	0x10
	incr R1	0x3100	0x12
	store R1, 0xA2	0x21A2	0x14
	jmp xx	0x5026	0x16
yy:	decr R2	0x4200	0x18
	...	...	
	rsb	0x7000	0x24
xx:	zero R2	0x8200	0x26
	incr R2	0x3200	0x28
	store R2, 0xA0	0x22A0	0x2A
	jsr yy	0x6018	0x2C
	incr R2	0x3200	0x2E
	store R2, 0xA2	0x22A0	0x30
	...	...	

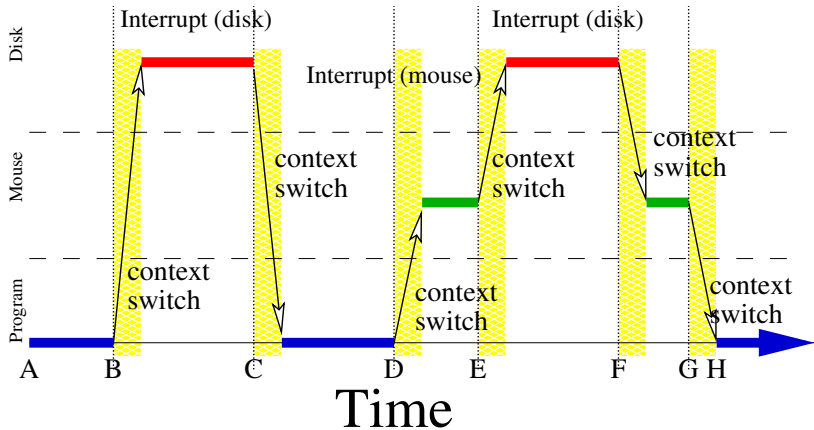
*Registers:* R0, R1, R2, R3, SP, PC

<i>Op Codes</i>	
load	0x1
store	0x2
incr	0x3
decr	0x4
jmp	0x5
jsr	0x6
rsb	0x7
zero	0x8



# Context Switch

- change from one running program to another “on the fly”
- need to be able to change back later
- to do this, we must save the “program state”
  - program counter
  - general purpose registers
  - status register
  - stack pointer
- each program must be resident in different parts of memory



- a hardware generated jump to subroutine (JSR) requested by an external hardware device (not the CPU)
- defined through the **interrupt vector**
  - resides somewhere in physical memory
  - is effectively an array of **function pointers**
- may be delayed (*i.e.*; disabled, possibly temporarily) by **masking off**

# Interrupt Steps

## Interrupt →

- save PC on stack
- switch CPU to kernel (supervisor) mode
- load PC from vector using hardware id (vector contains the start address of the interrupt service routine)
- save all registers
- continue execution

## Return from Interrupt (RTI)



- restore registers
- restore saved PC value
- set CPU to previous mode

# Critical Sections

- a **critical section** contains reference(s) to data shared between two or more “threads of control”
  - one may be an **interrupt**
  - this may be two or more **processes**
  - this may be two or more **threads**
- must manage the data so all parties can rely on its value
  - if we fail to do this, data may be wrong for one party, or even corrupted for all

---

		load	R1, #i
i++;	↔	incr	R1
		store	R1, #i

# A Critical Example

```
c = 'A';  
for (i = 0; i < MAX; i++) {  
    a[i] = c++;  
    if (c > 'E')  
        c = 'B';  
}
```

Interrupt Service Routine

```
set_to_a() {  
    c = 'A';  
}
```

# Modern (UNIX-type) systems: 3 types of device drivers

**BLOCK** devices that read/write blocks (disc, tapes, SD-cards, CD-ROM/DVD, ...)

**NETWORK** local area network (Ethernet, token ring), remote connection network interface (PPP, ISDN, ...)

**CHARACTER** everything that does not fit the above two cases (serial ports, display/keyboard, mouse, joystick, touchscreen, ...)

# Device Driver Functions

**Top Half:** communicate with user, kernel

**probe** check for hardware at boot time

**open** called when a process does an open of a **device file** (e.g., `/dev/mouse`, `/dev/disk0`, `/dev/null` etc.)

- mostly initializes variables and maybe hardware
- generates a file descriptor in user space

**close** called when a process does a `close()` of the file descriptor created from `open` (usually just marks the device closed)

**read** called when a process does a `read(fd)` — transfers data from device to process

**write** called when a process does a `write(fd)` — queues data for device to output

**Bottom Half:** event handling

**interrupt service routine**

called when device generates an interrupt and does the I/O on the hardware



```

char get_keyb()
{
    while (???) {
    }

    return ???;
}

void keyb_intr()
{
    /* "magic" addresses for memory mapped I/O */
    char *keyb_reg = (char *) 0x2ff4200;
    char *keyb_status = (char *) 0x2ff4201;

    while (*keyb_status)
        ;

    ??? = *keyb_reg;
}

```

# Circular Queue (Ring Buffer)

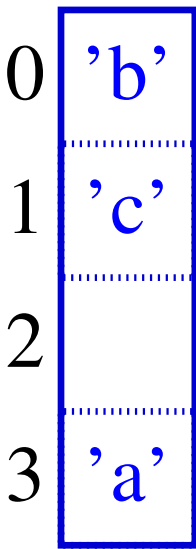
```
char Q[QSIZE];  
int numQ, startQpos, endQpos;  
numQ = startQpos = endQpos = 0;
```

Add into queue

```
Q[endQpos] = value;  
endQpos = (endQpos + 1) % QSIZE;  
numQ++;
```

Remove from queue

```
value = Q[startQpos];  
startQpos = (startQpos + 1) % QSIZE;  
numQ--;
```



# Critical Sections in Device Driver

- interrupts can be masked off:

`int state = spltty()` masks interrupt(s) to prevent a second interrupt from over-writing the first

`int splx(int prevState)` returns the interrupt mask to the previous state

- process can be “put to sleep” until there is space in a buffer:

`int tsleep(void *id, int flags, char *mesg, int timeo)`  
causes the current **thread** (program counter) to be **BLOCKED** until someone calls `wakeup()`

`int wakeup(void *id)` un-blocks *all* **threads** which are asleep and have the given **identifier**

## Device Driver Skeleton – read

```
static char Q[QSIZE];
static int numQ = 0, startQpos = 0, endQpos = 0;
devread(char *ubuf) { /** read one character */
    int s;
    s = spltty();
    while (numQ == 0) {
        splx(s); /** can't sleep inside critical section */
        tsleep(devread, ...); /* sleep */
        s = spltty();
    }
    *ubuf = Q[ startQpos ];
    startQpos = (startQpos + 1) % QSIZE;
    numQ--;
    splx(s);
}
```

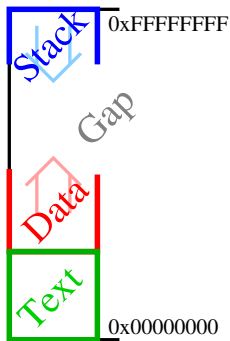
```
devintr() {  
    int s;  
    s = spltty();  
    if (QSIZE == numQ) {  
        /* queue is full ... */  
        splx(s); /* leave crit section before return */  
        return;  
    }  
    Q[ endQpos ] = GET_CHAR_FROM_DEVICE();  
    endQpos = (endQpos + 1) % QSIZE;  
    numQ++;  
    splx(s);  
    wakeup(devread);  
}
```

# Running Programs

The **execution** of a **program** (an “image”) is a **process**.  
The environment that the O/S provides for the execution of a program (*i.e.*; a process) is that of a **simplified** pseudo-computer.

UNIX/Mach process environment:

- **virtual** (simplified) address space
- **process context**
- no direct access to **I/O ports**



# Process Table Entry (Process Context)

- the **process table** contains an entry for each current process
  - if a process has more than one **thread** there is usually one **process table entry** per thread (`ps(1)` may or may not show it this way)
- each **entry** holds all the information about a process:
  - process (thread) context
    - program counter
    - registers (general and special purpose)
    - stack pointer
    - virtual address space reference
  - scheduling information
    - **BLOCKED** -vs- **RUNNABLE** -vs- **RUNNING**
    - priority
    - time used

# Process Status Listing

% ps auxw

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	1368	84	?	S	Jan11	0:04	init [3]
root	2	0.0	0.0	0	0	?	SW	Jan11	0:00	[keventd]
root	4	0.0	0.0	0	0	?	SWN	Jan11	0:00	[ksoftirqd_CPU0]
root	9	0.0	0.0	0	0	?	SW	Jan11	0:00	[bdflood]
root	5	0.0	0.0	0	0	?	SW	Jan11	0:00	[kswapd]
root	1609	0.0	0.0	1368	144	?	S	Jan11	0:00	klogd -x
root	1761	0.0	0.0	3508	480	?	S	Jan11	0:00	/usr/sbin/sshd
root	1920	0.0	0.2	2320	1380	?	S	Jan11	0:00	[login]
root	1921	0.0	0.0	1352	60	tty2	S	Jan11	0:00	/sbin/mingetty
andrew	8506	0.0	0.2	4692	1448	tty1	S	09:33	0:00	-csh
andrew	9373	0.0	0.5	8196	2980	pts/7	S	10:43	0:03	vim Processes.t
andrew	9795	0.0	0.2	4680	1428	pts/8	S	11:20	0:00	-bin/csh
andrew	10154	0.0	0.1	2728	780	pts/7	R	11:36	0:00	ps auxw



# Synchronization Primitives

Shared data can also be used between **processes/threads**

→ we need to manage critical sections at this level.

The two most popular **synchronization primitives** are:

**mutex:** bracket critical section so only 1 process allowed at a time:

```
critical_begin();  
... code in critical section ...  
critical_end();
```

**semaphores:** allow  $N$ -at-a-time

Co-operating sequential processes can be envisioned as a separate, dedicated processor for each – with no guarantee which runs first/faster etc.

# Mutex Synchronization: Producer

```
void add_queue(char aNewChar) {  
    int oldsize;  
  
    do { /** wait until not full */  
        critical_begin();  
        oldsize = numQ;  
        critical_end();  
    } while (oldsize == QSIZE);  
  
    critical_begin();  
    Q[ endQpos ] = aNewChar;  
    endQpos = (endQpos + 1) % QSIZE;  
    numQ++;  
    critical_end();
```

# Mutex Synchronization: Consumer

```
void del_queue(char aNewChar) {  
    int oldsize, tmpBuf;  
  
    do { /** wait until not empty */  
        critical_begin();  
        oldsize = numQ;  
        critical_end();  
    } while (oldsize == 0);  
  
    critical_begin();  
    tmpBuf = Q[ startQpos ];  
    startQpos = (startQpos + 1) % QSIZE;  
    numQ-;  
    critical_end();  
    return tmpBuf;  
}
```

# Implementing Mutual Exclusion (in Hardware)

```
static mutexFlag = 0;

critical_begin() {
    int oldFlag;

    do {
        /** does oldflag = flag , flag = 1 */
        TEST_AND_SET(&oldFlag, &mutexFlag);
        /** as one indivisible operation */
    } while (oldFlag);
}

critical_end() {
    mutexFlag = 0;
}
```

## Dekker's Algorithm (2 processes)

```
int need[2] = {0, 0};
int turn;

critical_begin(int who) {
    need[ who ] = 1;
    turn = !who;

    while (need[ !who ] && turn != who)
        ;
}

critical_end(int who) {
    need[ who ] = 0;
}
```

A **semaphore** is a synchronization primitive which has an internal integer counter supporting the following operations:

- signal** increments the internal counter

- wait** until the internal counter  $> 0$  ...*only then* decrement the counter

# Semaphore Implementation - Using Mutexes

```
SEM_WAIT(semaphore_t *s) {  
    semaphore_t oldsem;  
  
    /** wait until sem > 0 */  
    do {  
        critical_begin();  
        oldsem = *s;  
        if (*s > 0)  
            *s = *s - 1;  
        critical_end();  
    } while (oldsem == 0);  
}
```

```
SEM_SIGNAL(semaphore_t *s) {  
    critical_begin();  
    *s = *s + 1;  
    critical_end();  
}
```

- commercial/professional implementations use `tsleep()` to block the process instead of a busy-loop

# Semaphore Producer/Consumer

```
available_sem = QSIZE;  
used_sem = 0;  
critical_sem = 1;
```

```
void add_queue(char aNewChar) {  
    SEM_WAIT( &available_sem );  
    SEM_WAIT( &critical_sem );  
    Q[ endQpos ] = aNewChar;  
    endQpos = (endQpos + 1) % QSIZE;  
    SEM_SIGNAL( &critical_sem );  
    SEM_SIGNAL( &used_sem );  
}
```

```
void del_queue(char aNewChar) {  
    int tmpBuf;  
    SEM_WAIT( &used_sem );  
    SEM_WAIT( &critical_sem );  
  
    tmpBuf = Q[ startQpos ];  
    startQpos = (startQpos + 1)  
                % QSIZE;  
  
    SEM_SIGNAL( &critical_sem );  
    SEM_SIGNAL( &available_sem );  
  
    return tmpBuf;  
}
```



# Co-Operating Sequential Processes – Examples

```
semaphore_t U = 1;    semaphore_t D = 0;    int sharedch = 'a';
```

```
up() {  
    while (1) {  
  
        printf("U-%c ", sharedch);  
        if(++sharedch > 'z')  
            sharedch = 'a';  
  
    }  
}
```

```
down() {  
    while (1) {  
  
        printf("D-%c ", sharedch);  
        if(--sharedch < 'a')  
            sharedch = 'a';  
  
    }  
}
```

# Co-Operating Sequential Processes – Example #1

```
semaphore_t U = 1;    semaphore_t D = 0;    int sharedch = 'a';
```

```
up() {  
    while (1) {  
        SEM_WAIT( &U );  
  
        printf("U-%c ", sharedch);  
        if(++sharedch > 'z')  
            sharedch = 'a';  
        SEM_SIGNAL( &D );  
    }  
}
```

```
down() {  
    while (1) {  
        SEM_WAIT( &D );  
  
        printf("D-%c ", sharedch);  
        if(--sharedch < 'a')  
            sharedch = 'a';  
        SEM_SIGNAL( &U );  
    }  
}
```

Output: U-a D-b U-a D-b U-a ...

# Co-Operating Sequential Processes – Example #2

```
semaphore_t U = 2;    semaphore_t D = 0;    int sharedch = 'a';
```

```
up() {  
    while (1) {  
        SEM_WAIT( &U );  
  
        printf("U-%c ", sharedch);  
        if(++sharedch > 'z')  
            sharedch = 'a';  
        SEM_SIGNAL( &D );  
    }  
}
```

```
down() {  
    while (1) {  
        SEM_WAIT( &D );  
        SEM_WAIT( &D );  
        printf("D-%c ", sharedch);  
        if(--sharedch < 'a')  
            sharedch = 'a';  
        SEM_SIGNAL( &U );  
        SEM_SIGNAL( &U );  
    }  
}
```

Output: U-a U-b D-c U-c U-d D-e U-d U-e D-f U-e U-f D-g U-f ...

# Co-Operating Sequential Processes – Example #3

```
semaphore_t U = 0;    semaphore_t D = 2;    int sharedch = 'a';
```

```
up() {  
    while (1) {  
        SEM_WAIT( &U );  
        printf("U-%c ", sharedch);  
        if(++sharedch > 'z')  
            sharedch = 'a';  
        SEM_SIGNAL( &D );  
    }  
}
```

```
down() {  
    while (1) {  
        SEM_WAIT( &D );  
        SEM_WAIT( &D );  
        printf("D-%c ", sharedch);  
        if(--sharedch < 'a')  
            sharedch = 'a';  
        SEM_SIGNAL( &U );  
        SEM_SIGNAL( &U );  
    }  
}
```

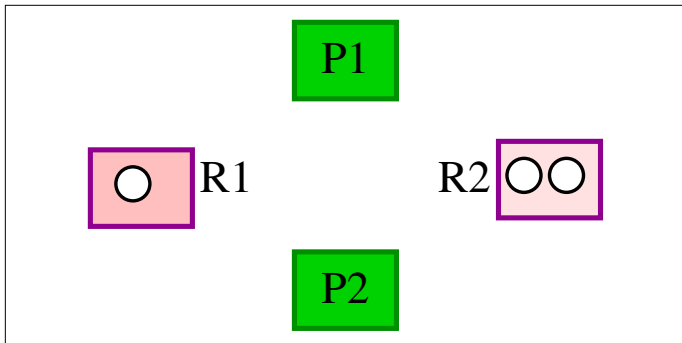
Output: D-a U-a U-b D-c U-c U-d D-e U-d U-e D-f U-e U-f D-g ...

# Semaphore Hints:

- all semaphores must be properly initialized
- always work from the definition of `wait_sem()` and `signal_sem()`
- normally, there will be a `wait_sem` with a corresponding `signal_sem()` on the same semaphore
- the order of `wait_sem()` operations is quite important; the order of **signal** less so

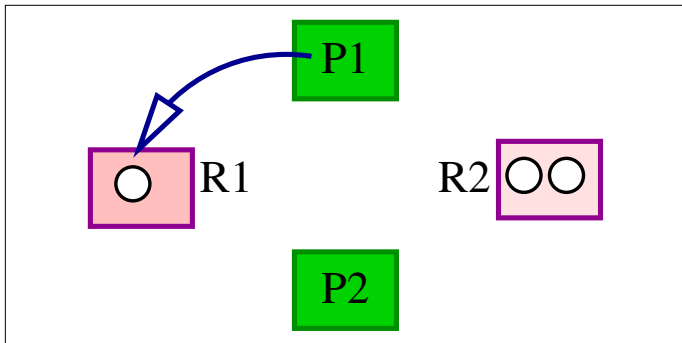
# Deadlock

- deadlock occurs when processes *hold* resources and they *require* resources that other processes already hold
- shown by the presence of a cycle in the request allocation graph



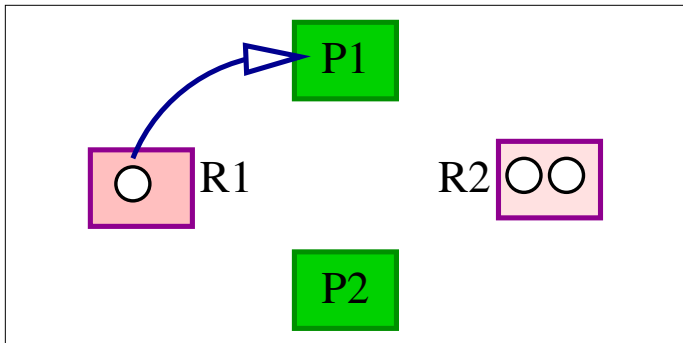
# Deadlock

- deadlock occurs when processes *hold* resources and they *require* resources that other processes already hold
- shown by the presence of a cycle in the request allocation graph



# Deadlock

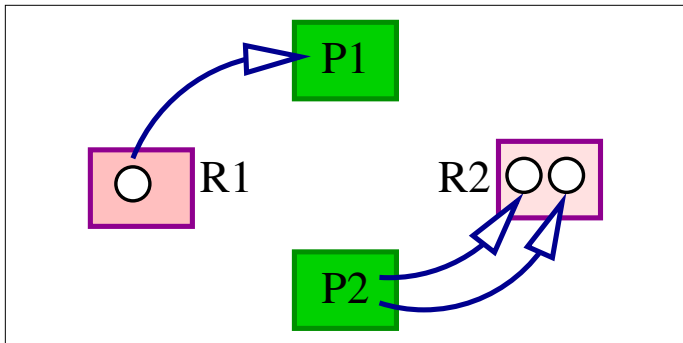
- deadlock occurs when processes *hold* resources and they *require* resources that other processes already hold
- shown by the presence of a cycle in the request allocation graph





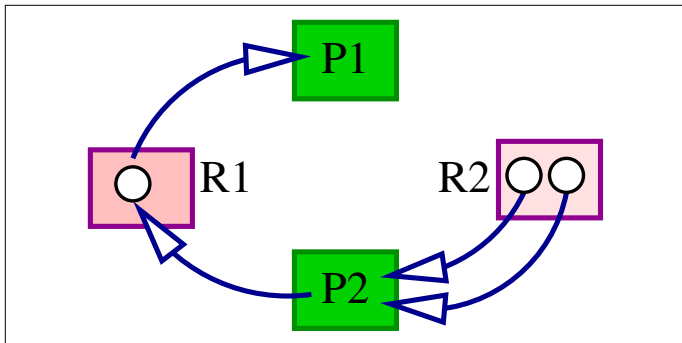
# Deadlock

- deadlock occurs when processes *hold* resources and they *require* resources that other processes already hold
- shown by the presence of a cycle in the request allocation graph



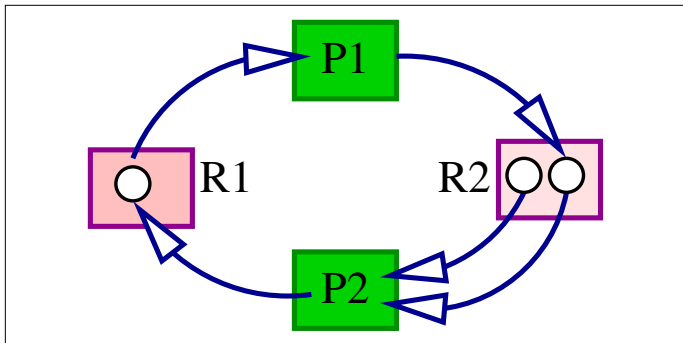
# Deadlock

- deadlock occurs when processes *hold* resources and they *require* resources that other processes already hold
- shown by the presence of a cycle in the request allocation graph



# Deadlock

- deadlock occurs when processes *hold* resources and they *require* resources that other processes already hold
- shown by the presence of a cycle in the request allocation graph



# Java `synchronized` keyword

- The `synchronized` keyword can be used on a whole method or a block to provide a critical section
- method locking:

```
public class SyncCount {  
    private int count = 0;  
    public synchronized void increment() { count++; };  
    public synchronized int getValue() { return count; };  
}
```

- block locking:

```
public void someMethod() {  
    while(keepGoing()) {  
        synchronized(someObject) {  
            count++;  
        }  
    }  
}
```

**Shared Memory** blocks of shared process image

**Semaphores/Mutexes** integer data values for flow control

**Signals** a software “interrupt”

**Pipes** an imitation file shared by two processes

- used to indicate that some important event has happened
- sequence of events
  - process A is running and performing some task
  - process B sends a signal to process A
  - process A interrupts whatever it is doing ...
  - call a signal handler (a function) ...
  - returns to whatever it was doing before
- there is no direct notification to process B that anything has been done

# Signal Values

Signals are simply integer valued actions, similar to interrupts. Some common signals are:

1 HUP	hangup	7	BUS	bus error
2 INT	interrupt	11	SEGV	segmentation violation
3 QUIT	quit	30	PWR	power failure
4 ILL	illegal instruction	15	TERM	terminate process
5 TRAP	trace trap	10	USR1	user defined #1
9 KILL	kill process	12	USR2	user defined #2

- The left column contains signals typically found at the indicated numbers.
- The right column contain common signals whose ID number may vary between platforms; the ID numbers for linux are provided.
- The numbers for a given platform may be found by running `kill -l`

## Signal Example

```
#include <signal.h>

void signalHandler(int code)
{
    fprintf(stderr,
              "Interrupt caught — cleaning up\n");
    performCleanup();
    exit(1);
}

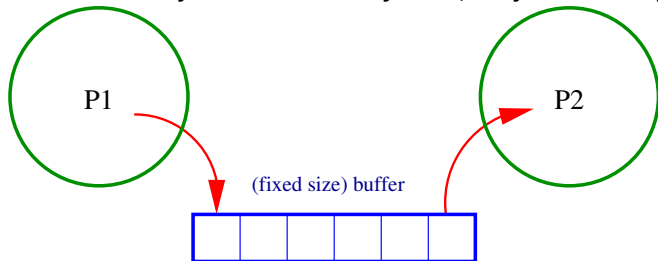
void main()
{
    signal(SIGINT, signalHandler);

    :
```



# Pipes

A **pipe** is a file descriptor pair shared between processes, attached to a buffer common to both processes. The buffer is not stored anywhere in a filesystem, only in memory.



Basic process scheduler loop:

- when a process can no longer run (exits, waiting for I/O or time-slice used up)
  - mark process RUNNABLE or blocked (and why – blocked status)
  - save process context in process table entry
- search process table for another RUNNABLE process
  - mark this process running
  - restore process context for this process
- the system now runs this process, picking up where it left off (just as with a hardware interrupt)

# Scheduling Algorithm or “Queuing Discipline”

## FIFO

- if jobs/processes/...repeatedly join the queue this is typically called “round robin.”
- typically, round robin is preemptive

## SJF

- shortest job first
- allow the job with the shortest running time to go next (jump to the head of the queue)
- not preemptive

---

A preemptive discipline permits a job to be preempted before it completes

# Scheduling Algorithms/Queueing Disciplines

- FIFO and round robin are generally considered to be **fair**, as jobs cannot be stuck waiting forever
- SJF is optimal w.r.t. throughput:
  - *iff* you always run the job that will finish in the shortest time.
  - will always finish more jobs in a given elapsed time than any other ordering
  - it is **unfair**, and impractical for processes (can be approximated)

# Scheduling Algorithms/Queuing Disciplines

In general:

- preemptive schedulers will perform better (*i.e.*; smoother response) than non-preemptive schedulers and are **required** for interactive systems
- schedulers will normally bias towards short jobs, such as updating an editor after a keystroke
- operators/users will be able to set/adjust priorities for jobs, see `nice(1)`
  - dynamic priority is updated during context switch based on last run time
- there will always be a trade-off between coarse -vs- fine granularity of the time-slices:
  - coarse** : low overhead/poor interactive response
  - fine** : higher overhead/better interactive response

# Scheduling Algorithms/Queueing Disciplines

- if preemption occurs extremely frequently such that each job gets an infinitesimally small time each time it runs, the discipline is called “processor sharing”
- whereas, if the preemption time is infinitely large, the discipline is called FIFO

# Scheduling Algorithms/Types of Tasks

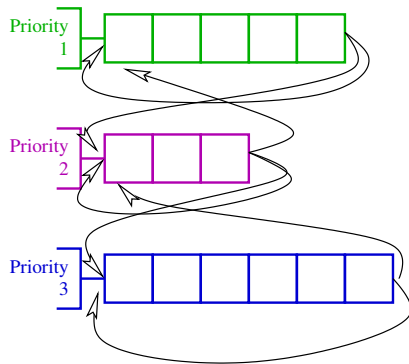
**batch** insensitive to gaps in runtime; get overall task done

**interactive** sensitive to runtime gaps; typically lots of I/O

**real-time** critically sensitive to runtime gaps

- demand fixed fraction of CPU/unit time

# Priority Queues (VMS/Windows)



Some systems vary the priority dynamically, others do not