We will start at 9:05

Process Control Blocks

PCBs

• Where the OS can find all the information it needs to know about a process.
    - memory
    - open streams/files
    - devices, including abstract ones like windows
    - links to condition handlers (signals)
    - processor registers (single thread)
    - process identification
    - process state - including waiting information
    - priority
    - owner
    - which processor
    - links to other processes (parent, children)
    - process group
    - resource limits/usage
    - access rights
    - process result - may be waited for by another process
- Doesn't have to be kept together
  Different information is required at different times

• UNIX for example has two separate places in memory with this information. One of them is in the kernel the other is in user space. Windows does the same.

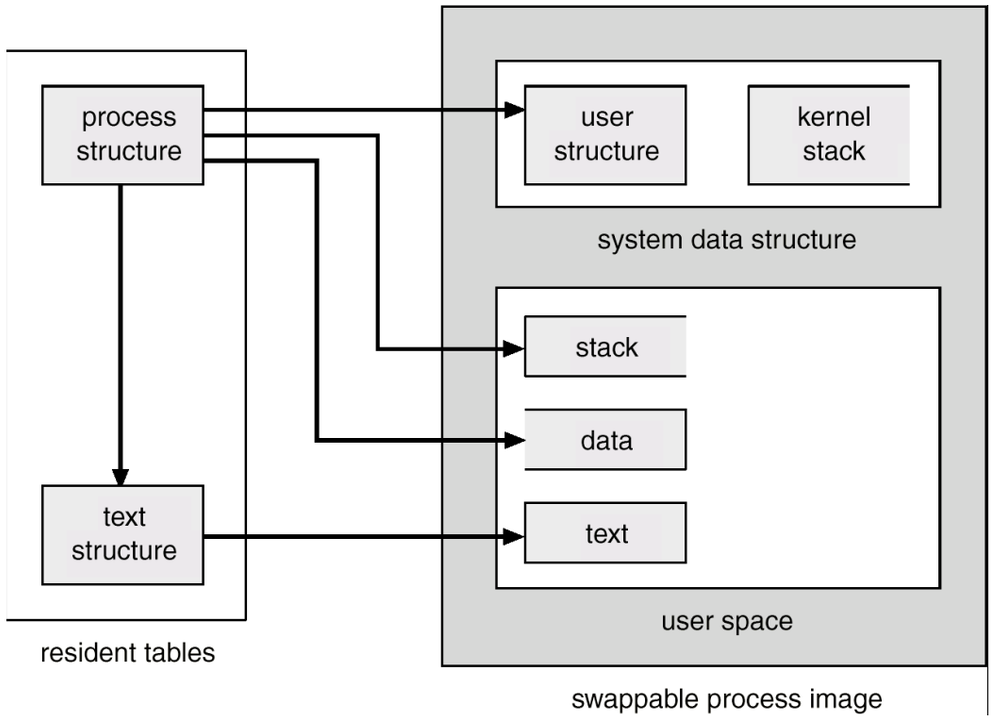
Linux PCBs

- In the same order as the previous slide.

```
/* memory management info */
    struct mm_struct *mm;
/* open file information */
    struct files_struct *files;
/* tss for this task */
    struct thread_struct tss;
  int pid;
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  long priority;
  unsigned short uid,euid,suid,fsuid;
#ifdef __SMP__
    int processor;
  #endif
struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
/* limits */
    struct rlimit rlim[RLIM_NLIMITS];
    long utime, stime, cutime, cstime, start_time;
```

UNIX process parts

- The PCB is the box labelled **process structure** but the **user structure** maintains some of the information as well (only required when the process is resident).



Windows NT PCBs

Information is scattered in a variety of objects.

- Executive Process Block (EPROCESS) includes
  - KPROCESS and PEB (Process Environment Block)
  - pid and ppid (the ppid is not visible to Win32)
  - file name of program
  - window station - the screen or remote terminal
  - exit status
  - create and exit times
  - links to next process
  - memory quotas
  - memory management info
  - Ports for exceptions and debugging
  - Security information

NT PCB (cont.)

Kernel Process Block (**KPROCESS**) includes info the kernel needs to *schedule threads*

- Kernel and user times.
- Pointers to threads.
- Priority information.
- Process state
- Processor affinity

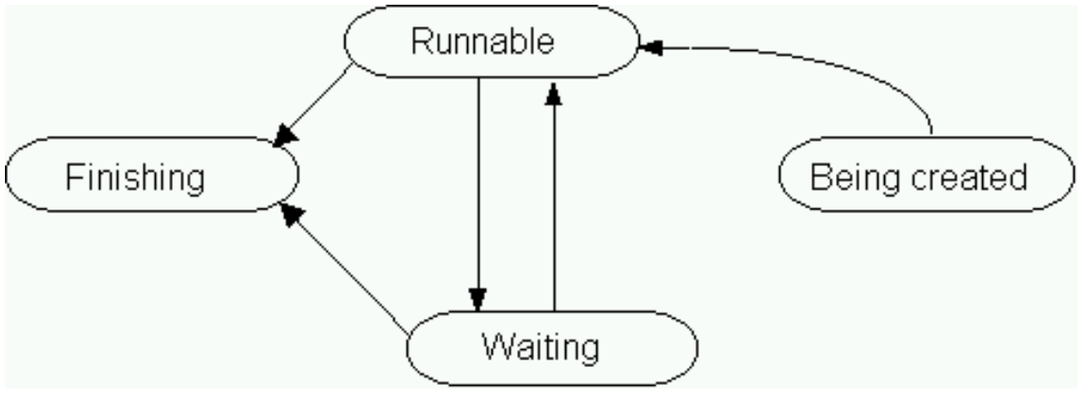
Process Environment Block (**PEB**) includes info which needs to be writable in user mode

- image info: base address, version numbers, module list
- heaps (blocks of one or more pages)

Process table and Thread structures

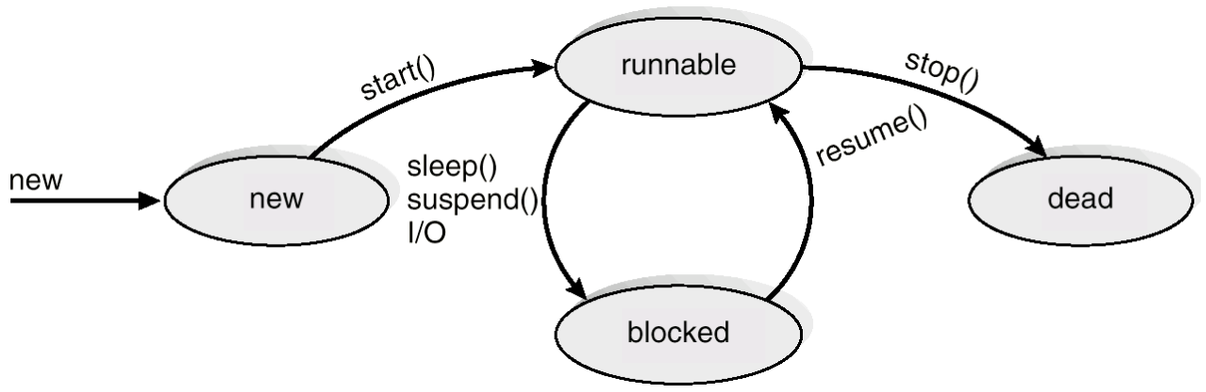
- Process Table
  - A collection of PCBs - originally an array
  - now a dynamic collection of pointers to PCBs
- Thread structures (like PCBs)
  - private memory (runtime stack) and static storage for local variables
  - processor registers
  - thread identification
  - thread state - including waiting information
  - priority
  - processor
  - associated process
  - thread group
  - thread result - maybe waited for by another thread

Process/Thread states



- At its simplest a process is either running or it is not.

Java thread states



Being created

- Different methods of creating processes
  - create process system call - takes a program name or a stream with the program data
  - copy process system call - a strange way of doing it but is now very widespread thanks to UNIX
  - create a new terminal session

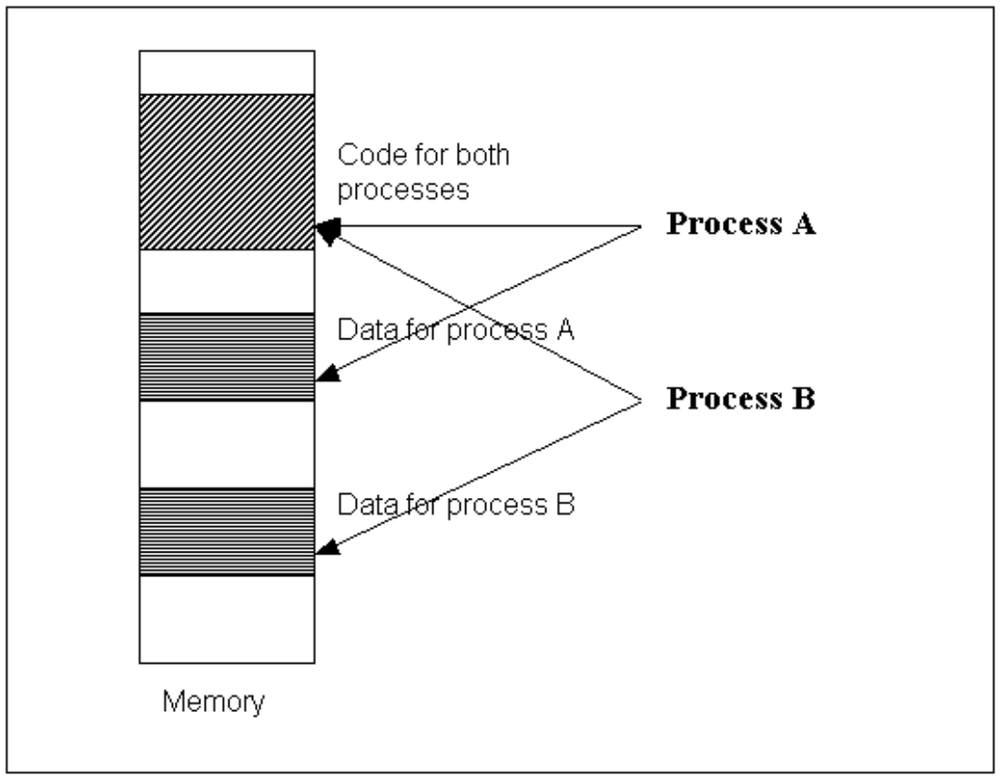
Being created (cont.)

Whichever way

- find a spare (or create a new) PCB (what if there isn't one?)
- mark it "being created"
- generate a unique identifier
- get some memory (what if there isn't any?) or
    - at least fill in the page table entries
    - set up PCB fields with initial values
    - set priority, resource limits
    - when all set up change the state to "runnable"
    - this could be done by inserting into a queue of runnable processes
- What about other resources?
- Some OSs carefully allocate resources before a process runs
  (this prevents deadlock later)
- Others leave these to the process to collect as it runs

fork

- The UNIX fork call duplicates the currently running process.
  - parent process - the one which made the call
  - child process - the new one
- Traditionally memory was duplicated - the code was shared even from earliest days.
  - Share open files as well.
  - Open file information blocks will have the count of processes using them increased by one.
  - And shared memory regions.
  - Fork returns 0 in the child process and the child's pid in the parent.



Test

- How many times is the ps command executed?

```
#include <unistd.h>
#include <stdlib.h>

int main() {
   int i = 0;

   while (i < 2) {
                fork();
                system("ps -o pid,ppid,comm,stat");
                i++;
   }
}
```

- Usually calls to fork are followed by calls to exec in the child process.

```
if  (fork() == 0)
    execl("nextprog", …);
```

- This keeps the process the same (what does that mean?) but changes the program.

Unix Pipes (out of order for A1)

- Data gets put into the pipe and taken out the other end
  - implies buffering mechanism
  - what size pipe?
  - what about concurrent use - can writes interleave? etc
- In UNIX it starts as a way for a process to talk to itself.

```
int myPipe[2]; pipe(myPipe);
```

- The system call returns two UNIX file descriptors.
- `myPipe[0]` to read, `myPipe[1]` to write

  e.g. `write(myPipe[1], data, length);`

Empty and full pipes

- Reading processes are blocked when pipes are empty
- Writing processes are blocked when pipes are full (65536 bytes on recent Linuxes)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    int my_pipe[2];

    if (pipe(my_pipe) == -1) {
        perror("Creating pipe");
        exit(EXIT_FAILURE);
    }
    int cpid = fork();
    if (cpid == -1) {
        perror("Creating child");
        exit(EXIT_FAILURE);
    }
    if (cpid != 0) { // the parent
        int size;
        read(my_pipe[0], &size, sizeof(size));
        for (int i = 0; i < size; i++) {
            read(my_pipe[1], &i, sizeof(int));
            printf("%d ", i);
        }
    } else { // the child
        int size = 1000;
        write(my_pipe[1], &size, sizeof(size));
        for (int i = 0; i < size; i++) {
            write(my_pipe[1], &i, sizeof(int));
        }
    }
}
```

exec

- checks to see if the file is executable
- saves any parameters in some system memory
- releases currently held memory
- loads the program
- moves the saved parameters into the stack space of the new program
- ready to run again
- Fork used to copy the data memory of the process.
  If the child is going to do an exec this is a waste of effort.
  Particularly bad with virtual memory.

Two solutions

1. copy on write

- No copy is made at first.
- The data pages of the parent process are set to read only.
- If a write occurs the resulting exception makes a copy of the page for the other process – both copies are then marked writable.

2. vfork (see the Linux man page)

- Trust the programmers to know what they are doing.

With vfork - parent process blocks until child finishes or calls exec.

- How many calls to ps are created by the earlier code if we could use vfork?

Copy on write is the predominant strategy.

- It is used in many situations to improve throughput.

NT process creation

- open .exe file and create a section object (actually quite complex because of the different subsystems)
- create NT process object
- Set up EPROCESS block
- create initial address space
- create KPROCESS block
- finish setting up address space - including mapping the section object
- adds process block to the end of the list of active processes
- set up PEB
- creates initial thread (initially suspended)
- Win32 subsystem is notified about the new process (includes the arrow and hourglass cursor)
- initial thread starts
- goes through more startup in the context of the new process - includes loading and initializing DLLs

Before next time

Read from the textbook

- 3.2 Process Scheduling
- 3.3 Operations on Processes