Tentamen

Systemnära programmering, 7.5hp
27 oktober 2017

- **Skrivtid**: 14 – 18
- **Hjälpmedel**: **EN** av följande böcker

  Bilting & Skansholm: "*Vägen till C* " ELLER
  J.R. Hanly & E.B. Koffman: "*Problem Solving and Program Design in C* " ELLER
  A. Kelley & I. Pohl: "*A Book on C* " ELLER
  Brian W. Kernighan & Dennis M. Ritchie: "*C Programming Language*" ELLER
  Brian W. Kernighan & Dennis M. Ritchie: "*Programmeringsspråket C*" ELLER
  Håkan Strömberg: "*C genom ett nyckelhål*"
- Maxpoäng: 40 (Gräns för 3: **20**p (50%), 4: **26**p (65%), 5: **32**p (80%))
- Börja varje uppgift på *nytt* blad och fyll i koden på *varje* blad.
- Inget besök i tentasalen.
  Vid tveksamheter – beskriv er tolkning.
- Tänk på att man kan få poäng även om man inte lyckas lösa hela uppgiften.

  Lycka Till!

Uppgift 1 (6 p)

Förklara kortfattat följande begrepp och dess användning:

- Zombie-process
- Atomär operation
- Reentrant function
- Nonblocking I/O
- Race condition
- FIFO (i denna kurs)

Uppgift 2 (4 p)

Givet följande kodavsnitt:

```
char a[] = "ABC";
char *b = "123";
char *c = malloc(6);
char *d = c + 2;
char *e = a + 3;
c = "ABC";
```

Rita en bild av datastrukturerna *a*, *b*, *c*, *d* och *e* (med innehåll), där man tydligt ser vad som allokerats och använd pilar för att markera hur pekarvärden är satta (efter kodsnutten).

Uppgift 3 (5 + 1 p)

```
#include <...>
static int g = 0;

int main(int argc, char *argv[])
{
    int var = 0;
    pid_t ret;

    while (var++ < 5)
        if ((ret = fork()) < 0) {
            perror("fork error");
            exit(1);
        } else if (ret == 0) {
            var+=1;
            g--;
        } else {
            g+=2;
            var+=2;
            if (waitpid(ret, NULL, 0) != ret) {
                perror("waitpid error");
                exit(1);
            }
        }
    printf("mypid = %d parentpid = %d ret = %d var = %d g = %d\n",
            getpid(), getppid(), ret, var, g);
    exit(0);
}
```

(a) Vad skrivs ut av programmet ovan? PID-nummer får ni hitta på, bara relationen mellan de olika numren framgår. (Rita gärna en figur.)

(b) Blir det någon skillnad om man plockar bort waitpid? Motivera ditt svar.

Uppgift 4 (3 p)

Förklara skillnaden mellan följande tre sätt att försöka "komma undan" en signal: 1) *ignorera* signalen, 2) *blockera* signalen och 3) *installera en tom signalhanterare* (dvs en funktion som gör return direkt, utan att göra något).

Uppgift 5 (5 p)

Beskriv vad som händer när följande kommando exekveras av csh (alltså inte vad kommandot utför). Ta med bland annat, Hur startas processer och program? När, och av vilken process skapas pipor. Hur hanteras omdirigering etc. Beskriv utförligt och rita gärna figurer.

```
% com1 input.txt | com2 -o out.dat | com1 > myfile.txt
```

Uppgift 6 (2 p)

Ge 4 fördelar med trådar (jämfört med processer).

Uppgift 7 (8 p)

Skriv ett program i C

```
myprg opt file1 [file2 ...] command
```

som tar en option, ett kommando och minst ett filnamn som argument (t ex `myprg -l myfile.c myfile.o ls`) och startar kommandot, med optionen och första filnamnet som argument, i en egen process. Sedan samma sak för den andra filen osv, så många gånger som det finns filnamn. Kommandot ska endast vara ett kommandonamn, alltså inte ett helt kommando med argument och pipes mm.
Programmet ska utföra ungefär samma sak som om man ger kommandoraden
`% command opt file1; command opt file2; ...`
(t ex `% ls -l myfile.c; ls -l myfile.o`),
dvs varje kommando får en egen process och de exekverar i tur och ordning och man får tillbaka promten först efter det att alla kommandon exekverat klart. Eftersom prototyper till de funktioner som behövs finns givna så tänk på att vara noga med argumenten. Skulle ni behöva använda någon annan funktion kan ni beskriva vilka argument som den har (ifall ni inte kommer ihåg exakt). (Kom ihåg att testa saker och ting. Funktionen `system()` och liknande får ej användas.)

Uppgift 8 (2 + 4 p)

(a) Nämn en likhet och olikhet mellan en semafor och mutex (från pthread).

(b) Betrakta följande synkroniseringsproblem:

> Vi har två typer av processer, konsument och producent, som delar en gemensam buffert av fix storlek. Producenten genererar data (ett i taget), lägger in det i bufferten, och börjar sedan om igen. Samtidigt konsumerar konsumenten data genom att ta data från bufferten (ett i taget) och sedan konsumera det, också det i en evig loop. Problemet, förutom vanlig synkronisering, är att se till att producenten inte lägger till data när bufferten är full och att konsumenten inte tar data från en tom buffert.

Din uppgift är att skriva kod för producenten och konsumenten. Använd vanliga, räknande semaforer för synkronisering och en C-liknande pseudokod för att lösa problemet så att inget data går förlorat. Din lösning ska klara godtyckligt antal producenter och konsumenter och måste säkerställa korrekt synkronisering och den ska vara deadlock-fri. Du måste också tydligt deklarera och initialisera eventuella variabler och semaforer. Du får också förutsätta att det finns funktioner för att skapa, konsumera, lägga till och ta bort data:

```
item produceItem(void)

void consumeItem(item x)

void putItemIntoBuffer(item x)

item removeItemFromBuffer(void)
```

Tips: Den vanligaste lösningen använder 3 semaforer.

Glöm ej att de *enda* operationerna som kan användas på en vanlig räknande semafor är `wait` och `signal`. Man kan dessutom initiera en semafor vid deklarationen.

Har din lösning några begränsningar?

Funktionsprototyper

int **execl**(const char **pathname*, const char **arg0*, ... `/* (char *) 0 */`);
`<unistd.h>`
Returns: -1 on error, no return on success

int **execle**(const char **pathname*, const char **arg0*, ... `/* (char *) 0 ,` char *const *envp*[] `*/`);
`<unistd.h>`
Returns: -1 on error, no return on success

int **execlp**(const char **filename*, const char **arg0*, ... `/* (char *) 0 */`);
`<unistd.h>`
Returns: -1 on error, no return on success

int **execv**(const char **pathname*, char *const *argv*[]);
`<unistd.h>`
Returns: -1 on error, no return on success

int **execve**(const char **pathname*, char *const *argv*[], char *const *envp*[]);
`<unistd.h>`
Returns: -1 on error, no return on success

int **execvp**(const char **filename*, char *const *argv*[]);
`<unistd.h>`
Returns: -1 on error, no return on success

void **exit**(int *status*);
`<stdlib.h>`
This function never returns

int **fclose**(FILE **fp*);
`<stdio.h>`
Returns: 0 if OK, -1 on error

int **feof**(FILE **fp*);
`<stdio.h>`
Returns: nonzero (true) if end of file on stream, 0 (false) otherwise

int **fgetc**(FILE **fp*);
`<stdio.h>`
Returns: next character if OK, `EOF` on end of file or error

int **fgets**(char **buf*, int *n*, FILE **fp*);
`<stdio.h>`
Returns: buf if OK, `NULL` on end of file or error

FILE ***fopen**(const char **filename*, const char **type*);
`<stdio.h>`
type: `"r", "w", "a", "r+", "w+", "a+"`
Returns: file pointer if OK, `NULL` on error

pid_t **fork**(void);
`<sys/types.h>`
`<unistd.h>`
Returns: 0 in child, process ID of child in parent, -1 on error

int **fprintf**(FILE **fp*, const char **format*, ...);
`<stdio.h>`
Returns: #characters output if OK, negative value if output error

int **fputc**(int *c*, FILE **fp*);
`<stdio.h>`
Returns: *c* if OK, `EOF` on error

int **fputs**(const char **str*, FILE **fp*);
`<stdio.h>`
Returns: nonnegative value if OK, `EOF` on error

void **free**(void **ptr*);
`<stdlib.h>`

int **fscanf**(FILE **fp*, const char **format*, ...);
`<stdio.h>`
Returns: #input items assigned, `EOF` if input error or `EOF` before any conversion

int **getc**(FILE **fp*);
`<stdio.h>`
Returns: next character if OK, `EOF` on end of file or error

int **getchar**(void);
`<stdio.h>`
Returns: next character if OK, `EOF` on end of file or error

void *__malloc__(size_t *size*);
`<stdlib.h>`
Returns: nonnull pointer if OK, `NULL` on error

int __pclose__(FILE **fp*);
`<stdlib.h>`
Returns: 0 if OK, -1 on error

void __perror__(const char **msg*);
`<stdio.h>`

int __pipe__(int *fildes[2]*);
`<unistd.h>`
Returns: 0 if OK, -1 on error

FILE *__popen__(const char **cmdstring*, const char **type*);
`<stdlib.h>`
type: `"r", "w"`
Returns: file pointer if OK, `NULL` on error

int __printf__(const char **format*, ...);
`<stdio.h>`
Returns: #characters output if OK, negative value if output error

int __putc__(int *c*, FILE **fp*);
`<stdio.h>`
Returns: *c* if OK, `EOF` on error

int __putchar__(int *c*);
`<stdio.h>`
Returns: *c* if OK, `EOF` on error

int __puts__(const char **str*);
`<stdio.h>`
Returns: nonnegative value if OK, `EOF` on error

int __scanf__(const char **format*, ...);
`<stdio.h>`
Returns: #input items assigned, `EOF` if input error or `EOF` before any conversion

int __system__(const char **cmdstring*);
`<stdlib.h>`
Returns: termination status of shell

pid_t __wait__(int **statloc*);
`<sys/types.h>`
`<sys/wait.h>`
Returns: process ID if OK, 0, or -1 on error

pid_t __waitpid__(pid_t *pid*, int **statloc*, int *options*);
`<sys/types.h>`
`<sys/wait.h>`
options: 0, WNOHANG, WUNTRACED
Returns: process ID if OK, 0, or -1 on error